Advanced Computer Communications Assignment 2
My Network Chat System

Jhi Morris (19173632) - 24/10/2020

Usage

The server does not support any user input once launched. The server can be executed like so: '**./server n m wait_time**', where n is the initial number of threads for handling client connections, m is the number of threads added to the threadpool when it reaches capacity, and wait_time is the number of seconds an idle client will remain before being timed out.

Example: '**./server 10 4 120**' to start the server with ten threads that will gain four additional threads if nine clients are connected concurrently; each of which will be disconnected if they do not send a command for more than 120 seconds.

The client can be executed like so: '**./client ip port**', where ip is the hostname or IP address of the server, (IP addresses must be in IPv4 dot-decimal notation, and port is the network port that the server is running at.

Example: '**./client 192.168.1.234 1234**' to connect to a server running at 192.168.1.234 on port 1234.

Example: '**./client localhost 52001**' to connect to a server running on the same machine as the client, on port 52001.

Once launched, commands can be input into the client. The following commands are accepted, along with their expected arguments and a usage description:

'JOIN username realname' where username and realname are the names you would like to join the server with.

'NICK nickname' where nickname is the new nickname you would like to change to.

'WHO' returns the list of users on the server.

'WHOIS target' returns information on a user on the server.

'TIME' returns the current system time.

'PRIVMSG target text' where target is the person you would like to send the text to privately.

'BCASTMSG text' where text is what you would like to broadcast to all users.

'QUIT text' where text is what you would like to broadcast to all users before disconnecting.

Mutual Exclusion & Thread Synchronization

The primary resources shared between each thread in the server are the ownedClients list and the newClientQueue list. The newClientQueue stores the sock descriptor of each client before it is claimed by a thread in the threadpool. Once claimed, the thread will generate an entry for the connection in the newClientQueue list. As multiple threads may access the newClientQueue list or the ownedClients list at the same time, each has a pthread_mutex object that each thread locks before accessing or modifying either list or its nodes.

The EstablishedConnection structs which the ownedClients list stores each contain information for the connection, a messageQueue list, a msgEventFd event_fd, and a mutex pthread_mutex object. Each thread first locks the mutex before accessing or modifying the fields of the EstablishedConnection struct. The msgEventFd signal is used by other threads to signal to the parent thread that messages have been added to the messageQueue list.

The messageQueue list contained within each EstablishedConnection struct has a mutex pthread_mutex object which each thread first locks before inserting or removing messages to the list.

No deadlock should occur on any server resource allocation as each thread must first lock access to the parent of a resource before locking access to that resource, and that no thread ever requires access to multiple top-level resources at a time. This means that once a thread has locked a resource, it will always have the ability to lock access to all other resources required by it before the point at which it next unlocks all mutex that it has locked.

The client makes use of two threads; one of which handles user input and the sending of messages to the server; the other of which handles the receiving, parsing, and printing of messages from the server. The only resource that they share is the recvThread error field; which is used by either thread to signal to the other that an error has occurred. As the integer is naturally aligned and it is always set without testing, no mutual exclusion is required.

Known Issues

- User input is interspersed with messages from the server. With more time, this could be corrected, however the priority was low and the work required for this is quite high.
- User input is not interrupted by the connection being closed. It may take an input or two for the program to close once the connection is closed.

Sample Input & Output

```
./client lab218-c02.cs.curtin.edu.au 52000
USER>bcastmsg Hello world
USER>SERVER: Info: You must JOIN before you can do that.
join jm Jhi Morris
USER>SERVER: JOIN jm - Jhi Morris - 134.7.45.157
time
USER>SERVER: The current server local time is 23:41:24
WHOis jm
USER>SERVER: Jhi Morris - 134.7.45.157
nick jhim
USER>SERVER: Your new nickname is jhim.
whois jm
USER>SERVER: Info: Matching user not found.
WHOIS jhim
USER>SERVER: Jhi Morris - 134.7.45.157
bcastmsg Hello world!!
USER>jhim: Hello world!!
who
USER>SERVER: jm Jhi Morris jhim 134.7.45.157 44s
privmsg jm Hello me!
USER>SERVER: Info: Matching user not found.
privmsg jhim Hello me!
USER>SERVER: PM sent.
PM from jhim: Hello me!
quit Bye, nobody!
USER>SERVER: You have been chatting for 78 seconds. Bye jhim!
NETWORK Info: Connection closed...
```

Source Code

common.h

```
/* common.h
*AUTHOR: Jhi Morris (19173632)
*MODIFIED: 19/10/2020
*PURPOSE: Header for common.c. Provides many structs and defines used by server and client
executables.
*/

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <signal.h>
#include <netdb.h>

//server-destined commands:
#define INVALID 0
#define JOIN 1 //username, realname
#define NICK 2 //nickname
#define WHO 3 // -
#define WHOIS 4 //target username
#define TIME 5 // -
#define PRIVMSG 6 //target username, message
#define BCASTMSG 7 //message
#define QUIT 8 //exit message
//client-destined commands:
```

```
#define RECVPRIV 9 //source username, message
#define RECVBCAST 10 //source username, message
#define RECVQUIT 11 //source username, message
#define SERVMSG 12 //error or info from server
#define RECVTERM 13 //error or info from server

typedef struct Message {
  uint8_t command;
  uint64_t length;
  char *body;
} Message;

extern const char* const commands[];
extern const size_t commandsLen;

int sendMessage(const Message msg, int sock);

int recieveMessage(Message* msg, int sock);

uint64_t findDelim(char *str, uint64_t maxLen);
```

---

common.c

```
/* common.c
*AUTHOR: Jhi Morris (19173632)
*MODIFIED: 19/10/2020
*PURPOSE: Provides functions used by both the client and server executables
*/

#include "common.h"

const char *const commands[] = {"join", "nick", "who", "whois", "time", "privmsg", "bcastmsg",
"quit"};
const size_t commandsLen = sizeof(commands) / sizeof(commands[0]);

/* sendMessage
*PURPOSE: Sends the contents of the Message struct over the socket connection. It returns 'true' if
an error occurs.
*INPUT: Message message, int sock descriptor
*OUTPUTS: int error occured (boolean)
*/
int sendMessage(Message msg, int sock)
{
  int error = false;

  if(msg.length == 0)
  {
    msg.length = 1;
    msg.body = calloc(1, sizeof(char));
    msg.body[0] = '\0';
  }

  if(send(sock, &(msg.command), sizeof(msg.command), 0))
  {
    if(send(sock, &(msg.length), sizeof(msg.length), 0))
    {
      if(!send(sock, msg.body, sizeof(char) * msg.length, 0))
      { //failed to send body
        error = true;
      }
    }
    else
    { //failed to send length
      error = true;
    }
  }
  else
  { //failed to send command
    error = true;
```

```
  }

  return !error;
}

/* recieveMessage
*PURPOSE: Recieves the contents of a message from the socket connection and writes it into the
Message struct at the pointer passed into it. It returns 'true' if an error occurs.
*INPUT: int sock descriptor
*OUTPUTS: int error occured (boolean), Message msg
*/
int recieveMessage(Message *msg, int sock)
{
  int error = false;

  if(recv(sock, &(msg->command), sizeof(msg->command), MSG_WAITALL) == sizeof(msg->command))
  {
    if(recv(sock, &(msg->length), sizeof(msg->length), MSG_WAITALL) == sizeof(msg->length))
    {
      msg->body = calloc(msg->length + 1, sizeof(char));

      if(recv(sock, msg->body, sizeof(char) * msg->length, MSG_WAITALL) == sizeof(char) *
msg->length)
      {
        msg->body[msg->length] = '\0'; //ensure null termination
        error = false;
      }
      else
      { //failed to get full body
        error = true;
      }
    }
    else
    { //failed to get length info
      error = true;
    }
  }
  else
  { //failed to get command info or connection closed
    error = true;
  }

  return !error;
}

/* findDelim
*PURPOSE: Searches the string, up to the max length, for '\2' and returns its offset. Returns -1 if
not found.
*INPUT: char* string to be searched, uint64_t max search width
*OUTPUTS: uint64_t delimiter offset
*/
uint64_t findDelim(char *str, uint64_t maxLen)
{
  uint64_t ret = 0;

  while(ret <= maxLen && str[ret] != '\2')
  {
    ret++;
  }

  if(ret == maxLen)
  {
    ret = -1;
  }

  return ret;
}
```

server.h

```
/* server.h
*AUTHOR: Jhi Morris (19173632)
*MODIFIED: 19/10/2020
*PURPOSE: Header for server.c
*/

#include "common.h"
#include "linkedlist.h"
#include <time.h>
#include <poll.h>
#include <errno.h>
#include <sys/eventfd.h>

#define SERVER_PORT 52000
#define MAX_BACKLOG 10

typedef struct ConnectionData
{
  long timeout;
  LinkedList ownedClients; //EstablishedConnection* type
  LinkedList newClientQueue; //int* (sock descriptor) type
  pthread_cond_t newClientSignal;
  int exit;
} ConnectionData;

typedef struct EstablishedConnection
{
  int hasJoined;
  time_t timeJoined;
  char username[11];
  char realName[21];
  char nickname[11];
  char hostname[256]; //stores ipv4 ip if no hostname found
  LinkedList messageQueue; //Message* type
  int msgEventFd; //only the owner thread will read from this
  pthread_mutex_t mutex;
} EstablishedConnection;

EstablishedConnection* findName(LinkedList *list, const char *name, int length);

void broadcastMessage(Message bcast, LinkedList *connList);

void server(const unsigned int threads, const unsigned int threadIncrement, const unsigned int
timeout, int sock);

void* clientThread(void* connectionData);

void joinc(Message *msgIn, Message *msgOut, LinkedList *connList, EstablishedConnection *con);

void nickc(Message *msgIn, Message *msgOut, LinkedList *connList, EstablishedConnection *con);

void whoc(Message *msgOut, LinkedList *connList);

void whoisc(Message *msgIn, Message *msgOut, LinkedList *connList);

void timec(Message *msgOut);

void privmsgc(Message *msgIn, Message *msgOut, LinkedList *connList, EstablishedConnection *con);

void bcastmsgc(Message *msgIn, Message *msgOut, LinkedList *connList, EstablishedConnection *con);

void quitc(Message *msgIn, Message *msgOut, LinkedList *connList, EstablishedConnection *con);
```

server.c

```
/* server.c
*AUTHOR: Jhi Morris (19173632)
*MODIFIED: 24/10/2020
*PURPOSE: Handles server communication and processing
*/

#include "server.h"

/* main
*PURPOSE: Reads in and validates the server parameters from the command line
*  arguments and binds to a socket for server().
*INPUT: argv[1] initial threads, argv[2] thread increment, argv[3] seconds of idle
*  before connection timeout.
*OUTPUTS: -
*/
int main(int argc, char *argv[])
{
  int error = false;
  int sock = -1;
  long port;
  long threads, threadIncrement, timeout;

  if(argc != 4)
  {
    printf("Invalid number of arguments.\n");
    error = true;
  }

  if(!error)
  {
    threads = strtol(argv[1], NULL, 10);

    if(!error && threads < 1)
    {
      printf("First argument (number of initial threads) must be an integer greater than zero.\n");
      error = true;
    }

    threadIncrement = strtol(argv[2], NULL, 10);

    if(!error && (threadIncrement < 1))
    {
      printf("Second argument (number of threads per increment) must be a positive integer.\n");
      error = true;
    }

    timeout = strtol(argv[3], NULL, 10);

    if(!error && (timeout < 0 || timeout > 120))
    {
      printf("Third argument (timeout time) must be a positive integer.\n");
      error = true;
    }
  }

  port = SERVER_PORT;
  //port = strtol(argv[4], NULL, 10);

  if(!error && (port < 1 || port > 65535))
  {
    printf("Forth argument (port) must be an integer between 1 and 65535, inclusive.\n");
    error = true;
  }

  if(!error && (sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    printf("Socket error.\n");
```

```
    error = true;
  }

  int val = 1;
  struct sockaddr_in ip;
  if(!error && setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val)) < 0)
  {
    printf("Sockopt error.\n");
    error = true;
  }
  else
  {
    //bind to ipv4
    memset(&ip, 0, sizeof(ip));
    ip.sin_family = AF_INET;
    ip.sin_port = htons(port);
    ip.sin_addr.s_addr = INADDR_ANY;
  }

  if(!error && bind(sock, (struct sockaddr*)&ip, sizeof(ip)) < 0)
  {
    printf("Failed to bind to the port. Is there already a service running at this port?\n");
    error = true;
  }

  if(!error && listen(sock, MAX_BACKLOG))
  {
    printf("Failed to listen on port.\n"); //no clue how it might reach this condition
    error = true;
  }

  if(!error)
  {
    printf("Starting server. . .\n");
    server(threads, threadIncrement, timeout, sock);
    printf("Server shutting down. . .\n");
  }
  else
  {
    printf("Expected usage: './server n m wait_time', where n is the initial "\
    "number of threads for handling client connections, m is the number of "\
    "threads added to the threadpool when it reaches capacity, and wait_time "\
    "is the number of seconds an idle client will remain before being timed out.\n"\
    "Example: './server 10 4 120' to start the server with ten threads that will "\
    "gain four additional threads if nine clients are connected concurrently; "\
    "each of which will be disconnected if they do not send a command for more than 120
seconds.\n");
  }

  if(sock != -1)
  {
    close(sock);
  }

  return !error;
}

/* server
*PURPOSE: Manages the threadpool and handles incoming connections to the socket.
*INPUT: unsigned int initial threads, unsigned int thread increment, unsigned int idle time limit,
int socket descriptor
*OUTPUTS: -
*/
void server(const unsigned int threads, const unsigned int threadIncrement, const unsigned int
timeout, int sock)
{
  ConnectionData *cData = calloc(1, sizeof(ConnectionData));
  cData->timeout = timeout;
  cData->exit = false;
  cData->ownedClients.head = NULL;
  cData->newClientQueue.head = NULL;
  pthread_mutex_init(&(cData->ownedClients.mutex), NULL);
  pthread_mutex_init(&(cData->newClientQueue.mutex), NULL);
  pthread_cond_init(&(cData->newClientSignal), NULL);
```

```
  LinkedList threadList; //pthread_t* type
  threadList.head = NULL;

  for(int i = 0; i < threads; i++)
  { //create initial threadpool
    pthread_t *thread = calloc(1, sizeof(pthread_t));
    pthread_create(thread, NULL, clientThread, (void*)cData);
    enqueue(thread, &threadList);
  }

  while(!(cData->exit))
  {
    int *sd = calloc(1, sizeof(int));

    if((*sd = accept(sock, NULL, NULL)) < 0)
    {
      printf("Connection error.\n");
      cData->exit = true;
    }
    else
    {
      pthread_mutex_lock(&(cData->newClientQueue.mutex));

      printf("SERVER Info: New connection.\n");
      enqueue(sd, &(cData->newClientQueue));

      pthread_mutex_lock(&(cData->ownedClients.mutex));

      if(length(&(threadList)) - length(&(cData->ownedClients)) - length(&(cData->newClientQueue))
<= 1)
      { //if idle threads minus available jobs is 1 or less
        printf("SERVER Info: Adding new threads to pool.\n");
        for(int i = 0; i < threadIncrement; i++)
        {
          pthread_t *thread = calloc(1, sizeof(pthread_t));
          pthread_create(thread, NULL, clientThread, (void*)cData);
          enqueue(thread, &threadList);
        }
      }

      pthread_cond_signal(&(cData->newClientSignal));
      pthread_mutex_unlock(&(cData->newClientQueue.mutex));
      pthread_mutex_unlock(&(cData->ownedClients.mutex));
    }
  }
}

/* clientThread
*PURPOSE: Thread function. Handles a connection with a client; sending and recieving messages
to/from it
*INPUT: ConnectionData* connection data
*OUTPUTS: -
*/
void* clientThread(void* connectionData)
{
  ConnectionData *cData = (ConnectionData*)connectionData;

  while(!(cData->exit))
  {
    int *client = NULL;
    while(client == NULL)
    { //reduces thundering herd
      pthread_mutex_lock(&(cData->newClientQueue.mutex));

      if(cData->newClientQueue.head == NULL)
      { //ensure there is a new client
        pthread_cond_wait(&(cData->newClientSignal), &(cData->newClientQueue.mutex));
      }

      client = dequeue(&(cData->newClientQueue));

      pthread_mutex_unlock(&(cData->newClientQueue.mutex));
    }
```

```
EstablishedConnection *estCon = calloc(1, sizeof(EstablishedConnection));
estCon->messageQueue.head = NULL;
estCon->timeJoined = time(NULL);
estCon->hasJoined = false;
pthread_mutex_init(&(estCon->messageQueue.mutex), NULL);
estCon->msgEventFd = eventfd(0, 0);

struct sockaddr_in addr;
int len = sizeof(struct sockaddr_in);
getsockname(*client, (struct sockaddr*)&addr, &len);
if(!getnameinfo((struct sockaddr*)&addr, len, estCon->hostname, 256, NULL, 0, 0))
{ //failed to resolve ip to hostname
  char *tmp = inet_ntoa(addr.sin_addr);
  strncpy(estCon->hostname, tmp, 16);
}

struct pollfd polld[2];
polld[0].events = POLLIN;
polld[0].revents = 0;
polld[0].fd = estCon->msgEventFd;
polld[1].events = POLLIN;
polld[1].revents = 0;
polld[1].fd = *client;
free(client);
pthread_mutex_lock(&(cData->ownedClients.mutex));

enqueue(estCon, &(cData->ownedClients));

pthread_mutex_unlock(&(cData->ownedClients.mutex));

int quit = false;
int lastMsgTime = time(NULL);
while(!quit)
{
  int remTime = lastMsgTime - time(NULL) + cData->timeout;
  switch(poll(polld, 2, remTime*1000))
  {
    case -1: //error
    printf("NETWORK Error: Failed to poll connection.\n");
    quit = true;
    break;
    case 0: //timeout
    printf("NETWORK Info: Connection timed out.\n");
    quit = true;
    break;
    default: //data or message available or connection died
    if(polld[0].revents > 0)
    { //messages to be sent
      eventfd_t buf;
      polld[0].revents = 0;
      eventfd_read(estCon->msgEventFd, &buf); //clear eventfd

      pthread_mutex_lock(&(estCon->messageQueue.mutex));

      while(estCon->messageQueue.head != NULL && !quit)
      {
        Message *msgOut = dequeue(&(estCon->messageQueue));

        if(!sendMessage(*msgOut, polld[1].fd))
        { //failed to send message
          quit = true;
          printf("NETWORK Error: Failed to send message.\n");
        }

        free(msgOut->body);
        free(msgOut);
      }

      pthread_mutex_unlock(&(estCon->messageQueue.mutex));
    }

    if(polld[1].revents > 0)
    { //message to be recieved
```

```
          Message msgIn;
          polld[1].revents = 0;
          if(recieveMessage(&msgIn, polld[1].fd))
          {
            lastMsgTime = time(NULL);
            Message *msgOut;
            msgOut = calloc(1, sizeof(Message));

            if(!estCon->hasJoined)
            {
              if(msgIn.command == JOIN)
              {
                joinc(&msgIn, msgOut, &(cData->ownedClients), estCon);
              }
              else
              {
                if(msgIn.command == QUIT)
                {
                  msgOut->command = RECVTERM;
                  char msgText[512];
                  snprintf(msgText, sizeof(msgText), "You have been connected without login for %ld
seconds. Bye.", time(NULL) - estCon->timeJoined);
                  msgOut->length = strnlen(msgText, sizeof(msgText));
                  msgOut->body = calloc(msgOut->length, sizeof(char));
                  strncpy(msgOut->body, msgText, msgOut->length);
                }
                else
                { //invalid command until JOINed
                  msgOut->command = SERVMSG;
                  char errorMsg[] = "Info: You must JOIN before you can do that.";
                  msgOut->length = sizeof(errorMsg);
                  msgOut->body = calloc(1, sizeof(errorMsg));
                  memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
                }
              }
            }
            else
            {
              switch(msgIn.command)
              { //note QUIT and JOIN handled seperately above
                case NICK: nickc(&msgIn, msgOut, &(cData->ownedClients), estCon);
                break;
                case WHO: whoc(msgOut, &(cData->ownedClients));
                break;
                case WHOIS: whoisc(&msgIn, msgOut, &(cData->ownedClients));
                break;
                case TIME: timec(msgOut);
                break;
                case PRIVMSG: privmsgc(&msgIn, msgOut, &(cData->ownedClients), estCon);
                break;
                case BCASTMSG: bcastmsgc(&msgIn, msgOut, &(cData->ownedClients), estCon);
                break;
                case QUIT: quitc(&msgIn, msgOut, &(cData->ownedClients), estCon);
                break;
                default: //invalid command
                quit = true;
                printf("NETWORK Info: Invalid connection dropped.\n");
                break;
              }
            }

            if(!quit)
            {
              if(msgOut->command != INVALID)
              {
                pthread_mutex_lock(&(estCon->mutex));
                pthread_mutex_lock(&(estCon->messageQueue.mutex));

                enqueue(msgOut, &(estCon->messageQueue));

                pthread_mutex_unlock(&(estCon->messageQueue.mutex));

                eventfd_write(estCon->msgEventFd, 1);
```

```
            pthread_mutex_unlock(&(estCon->mutex));
          }
          else
          {
            free(msgOut);
          }

          if(msgIn.command == QUIT)
          { //wrap-up
            pthread_mutex_lock(&(estCon->messageQueue.mutex));

            //send remaining messages
            while(estCon->messageQueue.head != NULL && !quit)
            {
              Message *msgOut = dequeue(&(estCon->messageQueue));

              if(!sendMessage(*msgOut, polld[1].fd))
              { //failed to send message
                quit = true;
                printf("NETWORK Error: Failed to send message to quitting user.\n");
              }

              free(msgOut->body);
              free(msgOut);
            }

            pthread_mutex_unlock(&(estCon->messageQueue.mutex));
            quit = true;
          }
        }
        else
        {
          free(msgOut);
        }
      }
      else
      { //failed to recieve valid message or connection died
        quit = true;
        printf("NETWORK Info: Invalid connection dropped.\n");
      }
    }
    break;
  }

  if(quit)
  {
    if(polld[1].fd != -1)
    {
      close(polld[1].fd);
    }
  }
}

pthread_mutex_lock(&(cData->ownedClients.mutex));

//clean up messageQueue
while(estCon->messageQueue.head != NULL)
{
  free(dequeue(&(estCon->messageQueue)));
}

pthread_mutex_destroy(&(estCon->messageQueue.mutex));

//remove estcon from ownedClients list
LinkedListNode *node = cData->ownedClients.head;

if(node != NULL && node->data == estCon)
{
  cData->ownedClients.head = node->next;
}
else
{ //not first element
  int found = false;
```

```
      if(node != NULL)
      { //may be null if list is empty. should never happen
        while(node->next != NULL && !found)
        {
          if(node->next->data == estCon)
          {
            found = true;
            node->next = node->next->next;
          }
          else
          {
            node = node->next;
          }
        }
      }

      if(!found)
      { //connection not found. should never happen
        printf("LOCAL: Critical error, failed to free connection memory.\n");
        cData->exit = true;
      }
    }

    //now no other thread can find estcon, ensure none are using it
    pthread_mutex_lock(&(estCon->mutex));
    pthread_mutex_unlock(&(estCon->mutex));
    pthread_mutex_destroy(&(estCon->mutex));

    free(estCon);

    pthread_mutex_unlock(&(cData->ownedClients.mutex));
  }

  return NULL;
}

/* findName
*PURPOSE: Finds the client with the nickname matching the input name and returns a pointer to it.
*INPUT: LinkedList* list of EstablishConnection*s, string name to find, int length of name
*OUTPUTS: EstablishedConnection* found name. Returns NULL if not found
*NOTES: Assumes mutex lock on list already attained.
*/
EstablishedConnection* findName(LinkedList *list, const char *name, int length)
{
  int found = false;
  LinkedListNode *node = list->head;

  while(node != NULL && !found)
  {
    pthread_mutex_lock(&(((EstablishedConnection*)node->data)->mutex));

    if(!strncmp(name, ((EstablishedConnection*)node->data)->nickname, length))
    {
      found = true;
    }

    pthread_mutex_unlock(&(((EstablishedConnection*)node->data)->mutex));

    if(!found)
    {
      node = node->next;
    }
  }

  return found ? (EstablishedConnection*)(node->data) : NULL; //returns NULL if not found
}

/* broadcastMessage
*PURPOSE: Clones and appends enqueues message to each client in the list.
*INPUT: Message to be broadcasted, LinkedList* list of EstablishConnection*s
*OUTPUTS: -
*NOTES: Assumes mutex lock on client list already attained.
*/
void broadcastMessage(Message bcast, LinkedList *connList)
```

```
{
  LinkedListNode *node = connList->head;

  while(node != NULL)
  {
    EstablishedConnection *target = (EstablishedConnection*)node->data;

    Message *clonedBcast = calloc(1, sizeof(Message));
    clonedBcast->command = bcast.command;
    clonedBcast->length = bcast.length;
    clonedBcast->body = calloc(bcast.length, sizeof(char));
    memcpy(clonedBcast->body, bcast.body, bcast.length*sizeof(char));

    pthread_mutex_lock(&(target->mutex));
    pthread_mutex_lock(&(target->messageQueue.mutex));

    enqueue(clonedBcast, &(target->messageQueue));

    pthread_mutex_unlock(&(target->messageQueue.mutex));

    eventfd_write(target->msgEventFd, 1);

    pthread_mutex_unlock(&(target->mutex));

    node = node->next;
  }
}

/*
* Below (until the end of the file) are the functions which handle client
* commands. All of them take pointers for an input message and an output
* message. Some also take a pointer to the connection list and the current connection.
* All must write to msgOut, though if it is not required for msgOut to be sent
* then the command field should be set to INVALID.
*/


/* joinc
*PURPOSE: Implements parsing and response to the JOIN command.
*INPUT: Message input message,  LinkedList* list of EstablishConnection*s, EstablishedConnection*
client.
*OUTPUTS: Message* direct response
*/
void joinc(Message *msgIn, Message *msgOut, LinkedList *connList, EstablishedConnection *con)
{
  uint64_t offset = findDelim(msgIn->body, msgIn->length);

  if(offset > 0 && offset < sizeof(con->username))
  {
    if(offset != msgIn->length && msgIn->length - offset < sizeof(con->realName))
    {
      pthread_mutex_lock(&(connList->mutex));

      if(findName(connList, msgIn->body, offset) == NULL)
      { //if no duplicate name
        pthread_mutex_unlock(&(connList->mutex));
        pthread_mutex_lock(&(con->mutex));

        strncpy(con->username, msgIn->body, offset);
        strncpy(con->nickname, msgIn->body, offset);
        strncpy(con->realName, msgIn->body+offset+1, msgIn->length-offset);


        con->hasJoined = true;

        Message bcast;
        bcast.command = SERVMSG;
        char msgText[512];
        snprintf(msgText, sizeof(msgText), "JOIN %s - %s - %s", con->nickname, con->realName,
con->hostname);
        bcast.length = strnlen(msgText, sizeof(msgText));
        bcast.body = calloc(bcast.length, sizeof(char));
        strncpy(bcast.body, msgText, bcast.length);
```

```
        pthread_mutex_unlock(&(con->mutex));
        pthread_mutex_lock(&(connList->mutex));

        broadcastMessage(bcast, connList);

        pthread_mutex_unlock(&(connList->mutex));

        msgOut->command = INVALID; //direct message back not required
      }
      else
      { //nickname or username already taken
        pthread_mutex_unlock(&(connList->mutex));
        msgOut->command = SERVMSG;
        char errorMsg[] = "Info: That name is already taken, please try again.";
        msgOut->length = sizeof(errorMsg);
        msgOut->body = calloc(1, sizeof(errorMsg));
        memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
      }
    }
    else
    { //invalid real name length
      msgOut->command = SERVMSG;
      char errorMsg[] = "Error: Real name must be between 1 and 20 letters long.";
      msgOut->length = sizeof(errorMsg);
      msgOut->body = calloc(1, sizeof(errorMsg));
      memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
    }
  }
  else
  { //invalid username length
    msgOut->command = SERVMSG;
    char errorMsg[] = "Error: Username must be between 1 and 10 letters long.";
    msgOut->length = sizeof(errorMsg);
    msgOut->body = calloc(1, sizeof(errorMsg));
    memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
  }
}

/* nickc
*PURPOSE: Implements parsing and response to the NICK command.
*INPUT: Message input message,  LinkedList* list of EstablishConnection*s, EstablishedConnection*
client.
*OUTPUTS: Message* direct response
*/
void nickc(Message *msgIn, Message *msgOut, LinkedList *connList, EstablishedConnection *con)
{
  if(msgIn->length > 0 && msgIn->length < 10)
  {
    pthread_mutex_lock(&(connList->mutex));

    if(findName(connList, msgIn->body, msgIn->length) == NULL)
    {
      pthread_mutex_unlock(&(connList->mutex));
      pthread_mutex_lock(&(con->mutex));

      strncpy(con->nickname, msgIn->body, msgIn->length);

      pthread_mutex_unlock(&(con->mutex));

      msgOut->command = SERVMSG;
      char msgText[512];
      snprintf(msgText, sizeof(msgText), "Your new nickname is %s.", con->nickname);
      msgOut->length = strnlen(msgText, sizeof(msgText));
      msgOut->body = calloc(msgOut->length, sizeof(char));
      strncpy(msgOut->body, msgText, msgOut->length);
    }
    else
    { //nickname or username already taken
      pthread_mutex_unlock(&(connList->mutex));
      msgOut->command = SERVMSG;
      char errorMsg[] = "Info: That name is already taken, please try again.";
      msgOut->length = sizeof(errorMsg);
      msgOut->body = calloc(1, sizeof(errorMsg));
      memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
```

```
    }
  }
  else
  { //invalid nickname length
    msgOut->command = SERVMSG;
    char errorMsg[] = "Error: Nickname must be between 1 and 10 letters long.";
    msgOut->length = sizeof(errorMsg);
    msgOut->body = calloc(1, sizeof(errorMsg));
    memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
  }
}

/* whoc
*PURPOSE: Implements response to the WHO command.
*INPUT: LinkedList* list of EstablishConnection*s
*OUTPUTS: Message* direct response
*/
void whoc(Message *msgOut, LinkedList *connList)
{
  pthread_mutex_lock(&(connList->mutex));

  LinkedListNode *node = connList->head;

  msgOut->command = SERVMSG;

  while(node != NULL)
  { //check body length required
    EstablishedConnection *estCon = (EstablishedConnection*)(node->data);
    if(estCon->hasJoined)
    {
      char buff[512]; //space for username, hostname, time, etc.
      snprintf(buff, sizeof(buff), "%s %s %s %s %lds\n", estCon->username, estCon->realName,
estCon->nickname, estCon->hostname, time(NULL) - estCon->timeJoined);

      msgOut->length += strlen(buff);
    }

    node = node->next;
  }

  msgOut->body = calloc(msgOut->length, sizeof(char));
  unsigned long offset = 0;
  node = connList->head;

  while(node != NULL)
  { //fill body
    EstablishedConnection *estCon = (EstablishedConnection*)(node->data);
    if(estCon->hasJoined)
    {
      char buff[512]; //space for username, hostname, time, etc.
      snprintf(buff, sizeof(buff), "%s %s %s %s %lds\n", estCon->username, estCon->realName,
estCon->nickname, estCon->hostname, time(NULL) - estCon->timeJoined);

      strncpy(msgOut->body + offset, buff, strnlen(buff, sizeof(buff)) - 1);
      offset += strnlen(buff, sizeof(buff)) - 1;
    }

    node = node->next;
  }

  pthread_mutex_unlock(&(connList->mutex));
}

/* whoisc
*PURPOSE: Implements parsing and response to the WHOIS command.
*INPUT: Message input message,  LinkedList* list of EstablishConnection*s
*OUTPUTS: Message* direct response
*/
void whoisc(Message *msgIn, Message *msgOut, LinkedList *connList)
{
  EstablishedConnection *user;
  if((user = findName(connList, msgIn->body, msgIn->length)) != NULL)
  {
    pthread_mutex_lock(&(user->mutex));
```

```
    char buff[512]; //space for realname & hostname, etc.
    snprintf(buff, sizeof(buff), "%s - %s", user->realName, user->hostname);

    pthread_mutex_unlock(&(user->mutex));

    msgOut->command = SERVMSG;
    msgOut->length = strnlen(buff, sizeof(buff));
    msgOut->body = calloc(msgOut->length, sizeof(char));
    strncpy(msgOut->body, buff, msgOut->length);
  }
  else
  { //user not found
    msgOut->command = SERVMSG;
    char errorMsg[] = "Info: Matching user not found.";
    msgOut->length = sizeof(errorMsg);
    msgOut->body = calloc(1, sizeof(errorMsg));
    memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
  }
}

/* timec
*PURPOSE: Implements response to the TIME command.
*INPUT: -
*OUTPUTS: Message* direct response
*/
void timec(Message *msgOut)
{
  time_t timenow = time(NULL);
  struct tm *locTime = localtime(&timenow);

  msgOut->command = SERVMSG;
  msgOut->length = 42;
  msgOut->body = calloc(42, sizeof(char)); //space for below message
  strftime(msgOut->body, msgOut->length, "The current server local time is %H:%M:%S.", locTime);
}

void privmsgc(Message *msgIn, Message *msgOut, LinkedList *connList, EstablishedConnection *con)
{
  EstablishedConnection *user;
  uint64_t offset = findDelim(msgIn->body, msgIn->length);
  if((user = findName(connList, msgIn->body, offset)) != NULL)
  {
    Message *privMsg = calloc(1, sizeof(Message));

    privMsg->command = RECVPRIV;
    privMsg->length = strnlen(con->nickname, sizeof(con->nickname)) + msgIn->length - offset + 1;
//+1 for '\2'
    privMsg->body = calloc(privMsg->length, sizeof(char));
    strncpy(privMsg->body, con->nickname, strnlen(con->nickname, sizeof(con->nickname)));
    strncpy(privMsg->body+strnlen(con->nickname, sizeof(con->nickname)), msgIn->body + offset,
strnlen(msgIn->body + offset, msgIn->length - offset));

    pthread_mutex_lock(&(user->mutex));
    pthread_mutex_lock(&(user->messageQueue.mutex));

    enqueue(privMsg, &(user->messageQueue));

    pthread_mutex_unlock(&(user->messageQueue.mutex));

    eventfd_write(user->msgEventFd, 1);

    pthread_mutex_unlock(&(user->mutex));

    msgOut->command = SERVMSG;
    char msgText[] = "PM sent.";
    msgOut->length = sizeof(msgText);
    msgOut->body = calloc(1, sizeof(msgText));
    strncpy(msgOut->body, msgText, sizeof(msgText));
  }
  else
  { //user not found
    msgOut->command = SERVMSG;
    char errorMsg[] = "Info: Matching user not found.";
```

```
    msgOut->length = sizeof(errorMsg);
    msgOut->body = calloc(1, sizeof(errorMsg));
    memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
  }
}

/* bcastmsgc
*PURPOSE: Implements parsing and response to the BCASTMSG command.
*INPUT: Message input message, LinkedList* list of EstablishConnection*s, EstablishedConnection*
client.
*OUTPUTS: Message* direct response
*/
void bcastmsgc(Message *msgIn, Message *msgOut, LinkedList *connList, EstablishedConnection *con)
{
  Message bcast;
  bcast.command = RECVBCAST;
  bcast.length = msgIn->length + strnlen(con->nickname, sizeof(con->nickname)) + 1; //+1 for space
for '\2'
  bcast.body = calloc(bcast.length, sizeof(char));
  strncpy(bcast.body, con->nickname, strnlen(con->nickname, sizeof(con->nickname)));
  bcast.body[strnlen(con->nickname, sizeof(con->nickname))] = '\2'; //delimiter
  strncpy(bcast.body+strnlen(con->nickname, sizeof(con->nickname))+1, msgIn->body, msgIn->length);

  pthread_mutex_lock(&(connList->mutex));

  broadcastMessage(bcast, connList);

  pthread_mutex_unlock(&(connList->mutex));

  msgOut->command = INVALID; //direct message back not required
}

/* quitc
*PURPOSE: Implements parsing and response to the QUIT command.
*INPUT: Message input message,  LinkedList* list of EstablishConnection*s, EstablishedConnection*
client.
*OUTPUTS: Message* direct response
*/
void quitc(Message *msgIn, Message *msgOut, LinkedList *connList, EstablishedConnection *con)
{
  Message notice;
  char msgText[512];
  notice.command = SERVMSG;
  snprintf(msgText, sizeof(msgText), "%s is no longer in our chatting session.", con->nickname);
  notice.length = strnlen(msgText, sizeof(msgText));
  notice.body = calloc(notice.length, sizeof(char));
  strncpy(notice.body, msgText, notice.length);

  Message bcast;
  bcast.command = RECVQUIT;
  bcast.length = msgIn->length + strnlen(con->nickname, sizeof(con->nickname)) + 1; //+1 for space
for '\2'
  bcast.body = calloc(bcast.length, sizeof(char));
  strncpy(bcast.body, con->nickname, strnlen(con->nickname, sizeof(con->nickname)));
  bcast.body[strnlen(con->nickname, sizeof(con->nickname))] = '\2'; //delimiter
  strncpy(bcast.body+strnlen(con->nickname, sizeof(con->nickname))+1, msgIn->body, msgIn->length);

  pthread_mutex_lock(&(connList->mutex));

  broadcastMessage(bcast, connList);
  broadcastMessage(notice, connList);

  pthread_mutex_unlock(&(connList->mutex));

  msgOut->command = RECVTERM;
  snprintf(msgText, sizeof(msgText), "You have been chatting for %ld seconds. Bye %s!", time(NULL) -
con->timeJoined, con->nickname);
  msgOut->length = strnlen(msgText, sizeof(msgText));
  msgOut->body = calloc(msgOut->length, sizeof(char));
  strncpy(msgOut->body, msgText, msgOut->length);
}
```

client.h

```
/* client.h
*AUTHOR: Jhi Morris (19173632)
*MODIFIED: 19/10/2020
*PURPOSE: Header for client.c
*/

#include "common.h"

#define MAXMSGLENGTH 256
#define MAXMSGLENGTHSTR "255"

typedef struct RecieveThread
{
  int sock;
  int error;
} RecieveThread;

void client(int sock, int *recvError);

void* recieveThread(void *recieveThread);
```

client.c

```
/* client.c
*AUTHOR: Jhi Morris (19173632)
*MODIFIED: 24/10/2020
*PURPOSE: Handles client user interface and communication
*/

#include "client.h"

/* main
*PURPOSE: Reads in and validates the connection parameters from the command line
*  arguments, then attempts to connect to the server using them.
*INPUT: argv[1] IPv4 address or hostname, argv[2] port.
*OUTPUTS: -
*/
int main(int argc, char *argv[])
{
  int error = false;
  int sock;
  struct sockaddr_in ip;
  struct hostent *hostname;
  long portTest;

  if(argc != 3)
  {
    printf("Invalid number of arguments.\n");
    error = true;
  }
  else
  {
    portTest = strtol(argv[2], NULL, 10); //used to check for negative or out of range values
    ip.sin_port = htons(strtol(argv[2], NULL, 10));
    hostname = gethostbyname(argv[1]);
  }

  if(!error && hostname != NULL)
  {
    ip.sin_addr = *((struct in_addr*)hostname->h_addr_list[0]);
    ip.sin_family = hostname->h_addrtype;
  }
```

```
  else
  { //not routable hostname
    error = true;
    if(!error && inet_pton(AF_INET, argv[1], &ip.sin_addr) > 0)
    { //check ip validity and type
      ip.sin_family = AF_INET;
    }
    else
    { //not valid IPv4 address
      printf("First argument is not a valid hostname or IP address. IP addresses must be in IPv4
dot-decimal notation.\n");
    }
  }

  if(!error && (sock = socket(ip.sin_family, SOCK_STREAM, 0)) < 0)
  { //check able to make socket for ip type
    printf("Socket error. Check network connectivity of this machine.\n");
    error = true;
  }

  if(!error && (portTest < 1 || portTest > 65535))
  { //check port validity
    printf("Second argument (port) must be an integer between 1 and 65535, inclusive.\n");
    error = true;
  }

  if(!error && connect(sock, (struct sockaddr*)&ip, sizeof(ip)) < 0)
  {
    printf("Connection error. Check network connectivity of this machine and the remote server.\n");
    error = true;
  }

  if(!error)
  {
    signal(SIGPIPE, SIG_IGN); //failed socket operations are handled as they occur

    RecieveThread recvData;
    recvData.sock = sock;
    recvData.error = false;

    pthread_t recvThread;
    pthread_create(&recvThread, NULL, recieveThread, &recvData);
    client(sock, &recvData.error);

    pthread_cancel(recvThread);
  }
  else
  {
    printf("Expected usage: './client ip port', where ip is the hostname or "\
    "IP address of the server, (IP addresses must be in IPv4 dot-decimal notation, "\
    "and port is the network port that the server is running at.\n"\
    "Example: './client 192.168.1.234 1234' to connect to a "\
    "server running at 192.168.1.234 on port 1234\n"\
    "Example: './client localhost 52001' to connect to a server running on "\
    "the same machine as the client, on port 52001.\n");
  }

  return !error;
}

/* client
*PURPOSE: Handles user input for server communication
*INPUT: int sock descriptor, int* recieve error flag
*OUTPUTS: -
*/
void client(int sock, int *recvError)
{
  Message msg;
  msg.body = NULL;
  int exit = false;
  int valid;
  char input[MAXMSGLENGTH]; //space for command or max-length message

  while(!exit && !(*recvError))
```

```
{
  valid = true;
  printf("USER>");
  if(scanf("%"MAXMSGLENGTHSTR"s", input) == 1) //handle command
  {
    for(int i = 0; i < strlen(input); i++)
    { //convert to lower case
      if(input[i] >= 'A' && input[i] <= 'Z')
      {
        input[i] += 'a' - 'A';
      }
    }

    int i = 0;
    int found = false;
    while(!found && i < commandsLen)
    { //compare to known command strings
      if(!strcmp(commands[i], input))
      {
        found = true;
      }

      i++; //even once found we increment, as the integers for commands starts at 1
    }

    if(found)
    {
      msg.command = i;

      switch(i)
      {
        case JOIN: //two arguments
        case PRIVMSG: //falls through
        if(scanf("%"MAXMSGLENGTHSTR"s", input) == 1)
        {
          char remainder[MAXMSGLENGTH];
          scanf(" "); //clear leading whitespace
          if(fgets(remainder, MAXMSGLENGTH, stdin) != NULL)
          {
            remainder[strlen(remainder)-1] = '\0'; //remove trailing newline from fgets

            msg.length = strlen(input) + strlen(remainder) + 1; //+1 for '\2'
            msg.body = calloc(msg.length + 1, sizeof(char)); //+1 for '\0'
            strcpy(msg.body, input);
            msg.body[strlen(input)] = '\2'; //delimits first and second arg
            strcpy(msg.body+strlen(input)+1, remainder);

            valid = true;
          }
          else
          { //no second argument to read
            valid = false;
            printf("LOCAL Error: %s requires more information.\n", commands[i-1]);
          }
        }
        else
        { //no first argument to read
          valid = false;
          printf("LOCAL Error: %s requires an argument.\n", commands[i-1]);
        }

        break;
        case NICK: //single argument
        case WHOIS: //falls through
        case BCASTMSG:
        case QUIT:
        if(fgets(input, MAXMSGLENGTH, stdin) != NULL)
        {
          input[strlen(input)-1] = '\0'; //remove trailing newline from fgets

          msg.length = strlen(input);
          msg.body = calloc(msg.length, sizeof(char)); //+1 for '\0', -1 to skip leading space
          strcpy(msg.body, input+1); //+1 to skip leading space
```

```
            valid = true;
          }
          else
          { //no argument to read
            valid = false;
            printf("LOCAL Error: %s requires more information.\n", commands[i-1]);
          }

          break;
          case WHO: //no arguments
          case TIME: //falls through
          msg.length = 0;
          if(msg.body != NULL)
          {
            free(msg.body);
            msg.body = NULL;
          }
          break;
        }
      }
      else
      { //unknown command
        valid = false;
        printf("LOCAL Error: Unknown command.\n");
      }
    }
    else
    { //no input
      valid = false;
      printf("LOCAL Error: Unknown command.\n");
    }

    fflush(stdin);

    if(valid && !(*recvError))
    {
      if(!sendMessage(msg, sock))
      { //if failed to send
        exit = true;
      }

      free(msg.body);
      msg.body = NULL;
    }
  }

  if(*recvError)
  {
    printf("NETWORK Info: Connection closed...\n");
  }
}

/* recieveThread
*PURPOSE: Parses and prints messages from the server
*INPUT: RecieveThread* thread data
*OUTPUTS: -
*/
void* recieveThread(void *recieveThread)
{
  RecieveThread *recv = (RecieveThread*)recieveThread;

  while(!(recv->error))
  {
    Message msg;

    if(recieveMessage(&msg, recv->sock))
    {
      uint64_t delimLen = findDelim(msg.body, msg.length);

      switch(msg.command)
      {
        case RECVPRIV:
        if(delimLen > 0)
        {
```

```
          printf("PM from %.*s: %.*s\n", delimLen, msg.body, msg.length-delimLen,
msg.body+delimLen);
        }
        else
        { //lacking second operand
          recv->error = true;
          printf("LOCAL: Recieved invalid server command, closing...\n");
        }
        break;
        case RECVBCAST:
        if(delimLen > 0)
        {
          printf("%.*s: %.*s\n", delimLen, msg.body, msg.length-delimLen, msg.body+delimLen);
        }
        else
        { //lacking second operand
          recv->error = true;
          printf("LOCAL: Recieved invalid server command, closing...\n");
        }
        break;
        case RECVQUIT:
        if(delimLen > 0)
        {
          printf("%*s's last message: %*s\n", delimLen, msg.body, msg.length - delimLen,
msg.body+delimLen);
        }
        else
        { //no quit message
          printf("There is no last message from %s!\n", msg.body);
        }
        break;
        case RECVTERM: //falls through
        recv->error = true; //time to shut down
        case SERVMSG:
        printf("SERVER: %.*s\n", msg.length, msg.body);
        break;
        default:
        //invalid command
        printf("LOCAL: Recieved invalid server command, closing...\n");
        recv->error = true;
        break;
      }

      free(msg.body);
    }
    else
    { //failed to recieve
      recv->error = true;
    }
  }
}
```
